Flow-noise en temps réel

Aymeric Augustin

ÉCOLE POLYTECHNIQUE
PROMOTION X2003
AUGUSTIN Aymeric

RAPPORT DE STAGE D'OPTION SCIENTIFIQUE

Flow-noise en temps réel

NON CONFIDENTIEL

Option :	Informatique
Champ de l'option :	Synthèse d'images
Directeur de l'option :	Gilles Dowek
Directeur de stage :	Fabrice Neyret
Dates du stage :	10 avril – 23 juin 2006
Adresse de l'organisme :	Evasion - Laboratoire GRAVIR INRIA Rhône-Alpes 655, avenue de l'Europe Montbonnot 38334 Saint Ismier Cedex

Résumé

Les textures procédurales, comme le bruit de Perlin, fournissent des images statiques de très grande qualité, indépendamment de la résolution. Nous cherchons à animer de telles textures en temps réel. Ainsi, nous pourrons améliorer l'apparence de fluides — en particulier les fluides naturels commes les nuages, le feu, l'eau, la lave, la boue, etc. — en ajoutant du bruit animé aux petites échelles.

Pour obtenir un résultat réaliste, nous devons donner l'impression d'écoulement et de tourbillonnement qui caractérise les fluides réels. L'*écoulement* est obtenu avec des textures advectées, liées à un simulateur de fluides. Le cœur de notre travail est la simulation des *tourbillons* dans le flux. Nous utilisons le *flow-noise* pour ajouter de la rotation à un bruit de Perlin. Nous mesurons localement la vorticité et exploitons la théorie de Kolmogorov pour déterminer la quantité d'animation dans le fluide à différentes échelles. Ceci garantit que le spectre de notre flow-noise est physiquement correct.

Le calcul du flow-noise est coûteux ; cependant, nous montrons comment on peut l'effectuer en temps réel avec les cartes graphiques actuelles. Nous soulignons quelques problèmes et quelques optimisations de notre algorithme spécifiques à la programmation sur carte graphique ; nous fournissons le code source complet et documenté de notre *pixel shader*.

Abstract

Procedural textures, such as Perlin noise, enable us to achieve high quality, resolution independant static effects. Our research aims at providing real-time animation for such noise. Thus, we can enhance the apperance of fluids — especially geophysical fluids such as clouds, fire, water, lava, mud, etc. — by adding animated noise at small scales.

To achieve a realistic appearance, we must give the feeling of the flowing and swirling that are caracteristic of real flows. *Flowing* is obtained with advected textures, linked to a fluid solver. The main part of our work is simulating the *swirling* of the flow. We use flow-noise to add rotation to Perlin noise. Using a local measure of vorticity and Kolmogorov theory, we determine the amount of animation in the fluid at different scales. This ensures that our flow-noise has a physically correct spectrum.

Computing flow-noise is quite intensive ; however, we show how real-time framerates can be achieved with current GPUs. We highlight a number of GPU-specific issues and optimisations for our algorithm, and we include complete source code and documentation for our pixel shader.

Table des matières

Introduction

Le rendu de fluides en images de synthèse est un problème rendu particulièrement difficile par la complexité de la physique sous-jacente ; pourtant, nombre d'effets spéciaux et de jeux vidéo en ont besoin. De nombreux articles, comme [Sta99, Sta01, Sta03] et [SRF05], ont proposé des approches par la mécanique des fluides (*e.g. Computational Fluid Dynamics*).

Cependant, ces méthodes souffrent de gros inconvénients. Leur complexité les rend impraticables à haute résolution ; le contrôle par l'artiste est extrêmement réduit ; les caractéristiques de certains fluides naturels comme la lave, les nuages ou la boue, sont souvent mal connues ; enfin, les simplifications opérées peuvent modifier fondamentalement la dynamique des très petites échelles. Par conséquent, même si ces méthodes reproduisent de manière satisfaisante le comportement des fluides à grande échelle, elles sont incapables de fournir les détails et la résolution auxquels le public s'attend de nos jours.

L'artiste préfère contrôler les détails par des textures. Dans le cas de fluides, les textures fixes semblent peu naturelles, car un fluide est animé à toutes les échelles, et il est même plus turbulent aux petites échelles. Se posent également des exigences contradictoires : déplacer la texture avec le fluide, en préservant la continuité spatio-temporelle et les propriétés statistiques, c'est à dire en contrôlant la déformation.

Nous cherchons donc une méthode pour amplifier le mouvement et les tourbillons du fluide aux petites échelles, de même qu'une texture standard amplifie les détails d'un objet. Nous nous orientons vers des textures procédurales, c'est à dire des textures définies par programme, pour résoudre ces problèmes. Si possible, notre algorithme devrait être applicable en temps réel : nous recherchons donc la simplicité et l'efficacité.

Nous nous plaçons dans le contexte d'un fluide dont l'animation à grande échelle est assurée par un mécanisme extérieur. On peut utiliser un simulateur de fluides, ou un pré-calcul dans le cas d'un régime continu. Nous utiliserons une version personnalisée du simulateur de fluides en temps réel proposé dans [Sta03], qui est léger et facilement adaptable à nos besoins.

Dans la partie 1, nous présenterons les textures procédurales, nous décrirons le bruit de Perlin — notre « brique élémentaire » et nous exposerons l'état de l'art sur notre problématique, en particulier les travaux de Fabrice Neyret ([PN01, Ney03]). Ensuite, nous expliquerons dans la partie 2 notre méthode pour calculer un bruit de Perlin animé (*e.g. flow-noise*) et comment nous contrôlons son spectre de rotation en utilisant le spectre de vorticité de Kolmogorov. Nous finirons en décrivant notre implémentation en temps réel sur carte graphique dans la partie 3.

1 État de l'art

1.1 Introduction aux textures procédurales

1.1.1 Concept

Afin d'augmenter le réalisme des images de synthèse, il est courant de recourir aux textures. On obtient ainsi des images d'une grande richesse visuelle, sans augmenter le coût de la modélisation. Il existe deux types de textures :

- les textures *images*, qui consistent à plaquer une image sur la géométrie de l'objet représenté. Les problèmes classiques consistent à déterminer comment placer la texture, comment traiter le sur-échantillonnage ou le sous-échantillonnage, et bien sûr à dessiner l'image de départ.
- les textures *procédurales*, qui utilisent une approche totalement différente : les propriétés d'un point sont déterminées algorithmiquement à partir de ses coordonnées spatiales. Au lieu de dessiner une image, on écrit un programme.

On pourra trouver une introduction détaillée aux textures procédurales dans [EMP+03]. D'une manière générale, elles donnent un liberté bien plus grande à l'artiste et elles sont peu sensibles à la résolution. Cependant, il est plus difficile d'obtenir un effet particulier et il est délicat prévoir avec précision le résultat. La figure 1 donne quelques exemples [Bev] de textures procédurales.

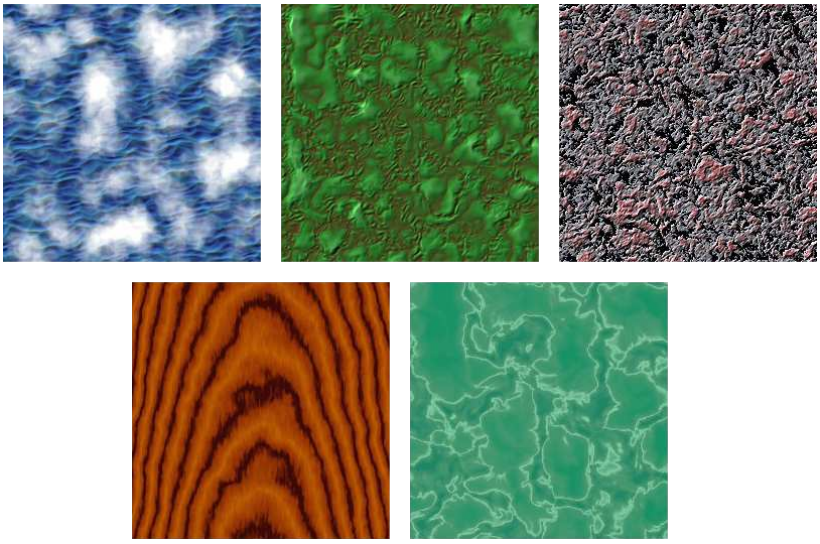

FIG. 1 – Exemples de textures procédurales

Les progrès des cartes graphiques ont permis la généralisation des *pixels shaders*, des programmes qui sont exécutés en chaque pixel affiché à l'écran pour calculer sa couleur : il s'agit typiquement d'une méthode procédurale. Nous utiliserons naturellement un *pixel shader* pour implémenter notre algorithme en temps réel. La différence entre textures images et procédurales n'est pas gommée pour autant : on peut habituellement déterminer si l'essentiel des propriétés d'un point (en particulier sa couleur de base) est plutôt définie par une image, ou plutôt par sa position spatiale.

1.1.2 Un exemple : le bruit de Perlin

L'exemple le plus classique de texture procédurale est le bruit de Perlin, décrit initialement par Ken Perlin en 1985 [Per85], puis dans une version améliorée en 2002 [Per02]. Le principe du bruit de Perlin est de définir une fonction par ses valeurs et ses gradients sur une grille, puis d'obtenir ses valeurs dans tout l'espace par interpolation. La fonction de bruit résultante est lisse et a un spectre contrôlé.

Version originale [Per85] fixe en chaque point à coordonnées entières (i, j, k) de l'espace quatre réels aléatoires indépendants, regroupés en un scalaire v_{ijk} et un vecteur $\overrightarrow{g_{ijk}}$. La fonction de bruit $\mathrm{Noise}(x, y, z)$ vérifie alors :

$$\forall (i,j,k) \in \mathbb{N}^3, \mathrm{Noise}(i,j,k) = v_{ijk} \text{ et } \overrightarrow{\nabla}\,\mathrm{Noise}(i,j,k) = \overrightarrow{g_{ijk}}$$

Les valeurs aux points de coordonnées non entières sont obtenues par interpolation lissée. Une interpolation cubique utiliserait $P_3(t) = 3t^2 - 2t^3$ comme interpolant, par exemple. La définition complète du bruit de Perlin est :

$$\begin{aligned}
&\forall (x,y,z) \in \mathbb{R}^3, \mathrm{Noise}(x,y,z) = \\
&\quad (1-P(\tilde{x}))(1-P(\tilde{y}))(1-P(\tilde{z}))\left(v_{\lfloor x\rfloor\lfloor y\rfloor\lfloor z\rfloor} + \overrightarrow{g_{\lfloor x\rfloor\lfloor y\rfloor\lfloor z\rfloor}}.\overrightarrow{(\tilde{x},\tilde{y},\tilde{z})}\right) \\
&+ (1-P(\tilde{x}))(1-P(\tilde{y}))(P(\tilde{z}))\left(v_{\lfloor x\rfloor\lfloor y\rfloor\lfloor z+1\rfloor} + \overrightarrow{g_{\lfloor x\rfloor\lfloor y\rfloor\lfloor z+1\rfloor}}.\overrightarrow{(\tilde{x},\tilde{y},\tilde{z}-1)}\right) \\
&+ (1-P(\tilde{x}))P(\tilde{y})(1-P(\tilde{z}))\left(v_{\lfloor x\rfloor\lfloor y+1\rfloor\lfloor z\rfloor} + \overrightarrow{g_{\lfloor x\rfloor\lfloor y+1\rfloor\lfloor z\rfloor}}.\overrightarrow{(\tilde{x},\tilde{y}-1,\tilde{z})}\right) \\
&+ (1-P(\tilde{x}))P(\tilde{y})P(\tilde{z})\left(v_{\lfloor x\rfloor\lfloor y+1\rfloor\lfloor z+1\rfloor} + \overrightarrow{g_{\lfloor x\rfloor\lfloor y+1\rfloor\lfloor z+1\rfloor}}.\overrightarrow{(\tilde{x},\tilde{y}-1,\tilde{z}-1)}\right) \\
&+ \cdots \\
&\qquad\qquad \text{où } \tilde{t} = t - \lfloor t\rfloor \text{ (partie décimale de } t).
\end{aligned}$$

On peut faire quelques remarques utiles pour la suite de l'exposé.

- Si P est un polynôme tel que $P(0) = 0$, $P(1) = 1$ et $P'(0) = P'(1) = 0$, alors Noise est continûment dérivable. C'est le cas de P_3.

– Cette méthode s'applique sans problème à n'importe quelle nombre de dimensions ; il suffit d'adapter le nombre de composantes du gradient et l'interpolation. Il est courant d'utiliser ce bruit en 2D pour des textures planes. Il est intéressant de l'obtenir en trois dimensions pour les *hyper-textures* [PH89], par exemple. On peut avoir besoin de quatre dimensions pour un bruit 3D animé dans le temps, si sa stochastique convient à la variation temporelle souhaitée.
– L'implémentation de référence [Perb] adopte une valeur nulle en tous les points de la grille, ce qui revient à fixer $v_{ijk} = 0$. L'intérêt est sans doute de simplifier l'algorithme et de mieux contrôler le spectre de bruit. Cependant, le risque d'artefacts est accru, car cela peut faire ressortir la grille des points à coordonnées entières. Cette modification facilite aussi l'interprétation mathématique dans le cadre de la théorie des ondelettes, que l'on exposera plus tard.

L'implémentation se fait en utilisant un certain nombre d'optimisations. Un jeu fini de vecteurs aléatoires est pré-calculé, et tous les gradients sont choisis dans cet ensemble. Un tableau de permutation aléatoire p également pré-calculé permet d'éviter les corrélations spatiales : au point (i, j, k), on utilise le vecteur numéro $p[p[i] + j] + k$, modulo le nombre de vecteurs. Le bruit devient périodique, mais en choisissant un jeu de départ suffisamment large — 64 vecteurs seulement peuvent suffire — cela ne se voit pas.

Perlin a publié sur le web une implémentation de référence en C pour une, deux ou trois dimensions [Perb].

Il existe de nombreuses manières de transformer ce bruit. La figure 2 illustre la manière la plus courante d'augmenter le détail du bruit : utiliser une somme fractale, définie par :

$$N(\vec{x}) = \sum_{i=0}^{m} 2^{-i} \,\mathrm{Noise}(2^i \vec{x})$$

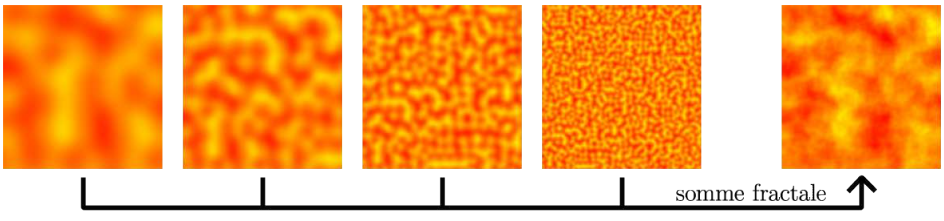

FIG. 2 – Bruit de Perlin et turbulence obtenue comme somme fractale

La figure 4 montre diverses variantes, correspondant aux fonctions du tableau 3.

$$N(\vec{x}) = \text{Noise}(\vec{x}) \qquad N(\vec{x}) = \sum_{i=0}^{m} 2^{-i}\,\text{Noise}(2^i\vec{x})$$

$$N(\vec{x}) = \sum_{i=0}^{m} 2^{-i}|\text{Noise}(2^i\vec{x})| \qquad N(\vec{x}) = 1 - \sum_{i=0}^{m} 2^{-i}|\text{Noise}(2^i\vec{x})|$$

FIG. 3 – Fonctions définissant les images de la figure 4

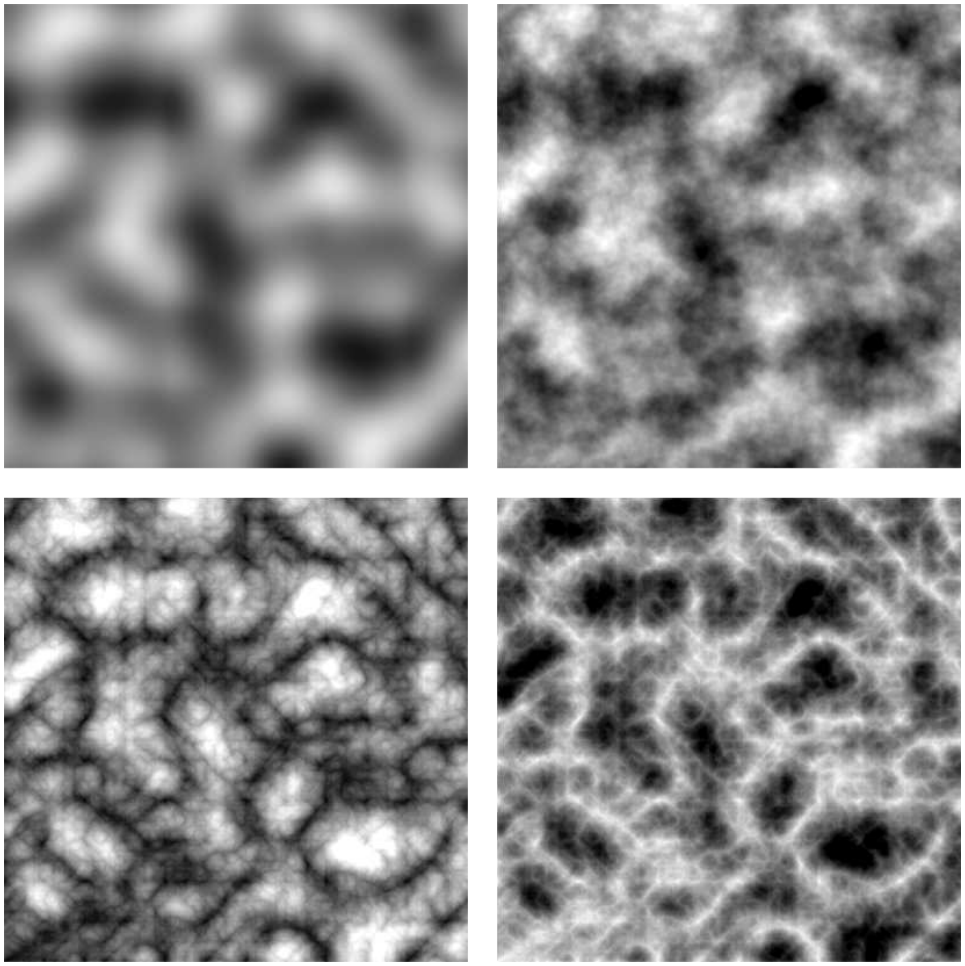

FIG. 4 – Différentes transformations du bruit

Perlin a consacré beaucoup de temps à créer des images basées sur cette fonction de bruit. Leur qualité et leur diversité sont extraordinaires compte-tenu des moyens informatiques de l'époque.

Version améliorée [Per02] propose trois améliorations principales à l'algorithme précédent. Au lieu de choisir les gradients dans un ensemble de vecteurs aléatoires, un jeu de 16 vecteurs explicitement définis est utilisé. Il contient les douze vecteurs qui vont du centre d'un cube au milieu de ces arêtes, plus quatre de ces vecteurs qui sont dupliqués pour obtenir un nombre puissance de 2. Le choix de ces vecteurs est prévu pour minimiser certains artefacts dus à l'anisotropie de la grille, car ils évitent les directions de ses arêtes et ses grandes diagonales. Ils ont également l'avantage de rendre les produits scalaires plus faciles à calculer.

Un polynôme interpolateur d'ordre 5 : $P_5(t) = 6t^5 - 15t^4 + 10t^3$ remplace le polynôme d'ordre 3 P_3. P_5 vérifie en plus $P_5''(0) = P_5''(1) = 0$, ce qui garantit que la fonction Noise est deux fois continûment dérivable et élimine ainsi certains artefacts, en particulier lorsque le bruit est utilisé pour déplacer une surface.

Enfin, [Per02] propose de fixer une permutation de $[\![0, 255]\!]$ et la manière de choisir le vecteur gradient au point (i, j, k) à partir de cette permutation, éliminant ainsi tout aspect aléatoire de l'algorithme. Cela rend possible une implémentation uniforme et déterministe de la fonction Noise.

Perlin a publié sur le web une implémentation de référence en Java de cet algorithme amélioré [Pera]. Elle est prévue uniquement pour fonctionner en trois dimensions.

1.2 Flow-noise

Une première solution pour appliquer une texture procédurale à un fluide à été proposée par Ken Perlin et Fabrice Neyret en 2001 dans [PN01] (également dans [EMP+03], pp. 384–389). L'article propose deux idées pour reproduire la turbulence d'un fluide : les gradients rotatifs et la pseudo-advection.

1.2.1 Gradients rotatifs

Le bruit de Perlin est défini par ses gradients sur une grille. Cependant, on peut aussi le visualiser comme une somme d'ondelettes, liées à chaque somment de la grille.

En effet, le gradient $\vec{g}$ en un point O de la grille a une influence sur les valeurs des points contenus dans les huit cubes dont O est sommet ; et cette influence de $\vec{g}$ décroît avec la distance à O, à cause de l'interpolation. Cette influence est représentée dans le cas bi-dimensionnel sur le deuxième graphe de la figure 5. Le produit de la fonction linéaire définie par le gradient et de

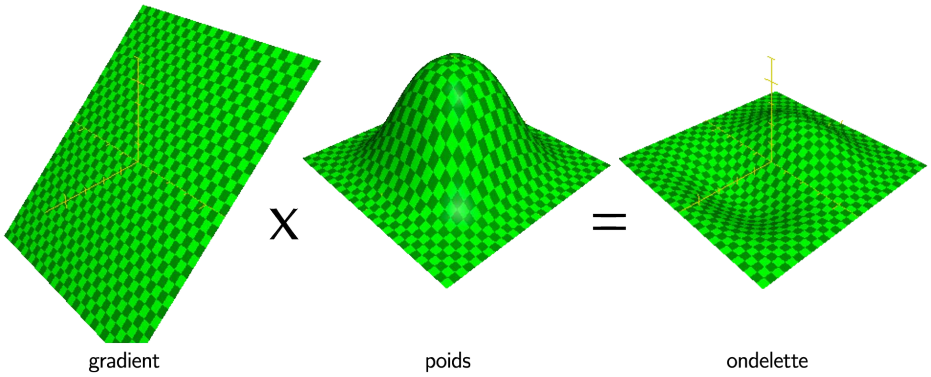

FIG. 5 – Générateur de bruit de Perlin

cette pondération est une ondelette. Le bruit de Perlin est le résultat de la somme des ondelettes placées en tous les points de la grille.

L'orientation du gradient détermine celle de l'ondelette ; ainsi, en donnant une rotation aux gradients, on fait tourner les ondelettes. C'est ce qu'on appelle le *flow-noise*. L'animation ainsi obtenue est par nature rotative. Autre avantage, à chaque instant, le bruit conserve sa structure. Nous reprenons cette idée, et donnons plus de détails en 2.1.

L'article évoque également une approche fractale ; dans ce cas, les petites échelles doivent être animées d'une rotation plus rapide pour obtenir un effet réaliste. Notons que la vitesse de rotation est uniforme spatialement, ce qui évite certains artefacts d'enroulement, dont nous reparlerons en détail en 2.1.2.

1.2.2 Pseudo-advection

Le bruit ainsi obtenu souffre d'un gros défaut lorsqu'on le plaque sur un fluide en mouvement : il ne suit pas le mouvement du fluide ; il « flotte » sur place en quelque sorte, sans être emporté par le courant.

La pseudo-advection apporte une solution partielle à ce problème, dans le cas d'un bruit fractal. Elle consiste à déplacer chaque échelle de bruit en fonction du « mouvement » généré par l'échelle supérieure. Ainsi, les petites échelles sont entraînées par les grandes échelles, simulant une forme d'advection au sein de la texture de bruit. L'article introduit un paramètre appelé « taux de régénération » pour contrôler l'activité du fluide. Ce nombre détermine le taux de pseudo-advection : plus il est élevé, plus l'advection est importante.

Nous n'utiliserons pas la pseudo-advection, pour deux raisons. Tout d'abord, nous mettons en place une vraie advection (cf. 1.3), ce qui atténue la nécessité de la pseudo-advection : l'expérience montre que le mouvement

général du fluide est largement prédominant par rapport à celui simulé par le bruit ajouté. De plus, la pseudo-advection devient passablement compliquée à déterminer, compte tenu de la manière de calculer le flow-noise que nous allons introduire ; et nous voulons limiter la complexité pour parvenir à une implémentation temps-réel.

1.3 Textures advectées

Lorsque le fluide a un mouvement global, la pseudo-advection ne suffit pas donner l'impression que le bruit suit ce mouvement ; un vrai mécanisme d'advection de la texture devient nécessaire. L'advection d'une grandeur consiste à déplacer cette grandeur avec le mouvement du fluide, c'est à dire à la lier au référentiel du fluide.

Le principe fondamental des textures advectées est d'advecter les coordonnées textures. Un mécanisme particulièrement élaboré a été proposé par Fabrice Neyret dans [Ney03]. Bien que nous utilisions une méthode plus simple, il est intéressant de comprendre cet article qui met en lumière les problématiques liées à l'advection. Nous nous plaçons ici dans un contexte bi-dimensionnel pour plus de simplicité, mais les résultats de l'article s'appliquent aussi en 3D.

1.3.1 Régénération et latence

Supposons que l'on place une grille sur la zone correspondant au fluide, et que la texture soit attachée à cette grille. On peut lier cette grille au fluide en déplaçant ses sommets suivant le mouvement du fluide, c'est à dire en les advectant [1]. Ainsi, la texture se déplace bien avec le fluide.

En pratique, à cause de l'étirement du fluide, la déformation de la texture devient vite très importante. Ses propriétés spatiales, comme la taille caractéristique et la forme de ses motifs, ou encore la répartition des couleurs sont fortement modifiées. Pour contrer ce problème, il faut régénérer la texture, ce qui revient à ré-initialiser les coordonnées texture.

Cette régénération va provoquer une brutale discontinuité temporelle, qu'il est indispensable de masquer. Pour cela, on mélange plusieurs couches de texture, dont les opacités varient au cours du temps. La texture i est régénérée lorsque son opacité α_i est nulle : par exemple on prend

$$\alpha_i = \frac{1}{2}\cos\left(2\pi\frac{t_i}{\lambda}\right)$$

Utiliser trois textures déphasées de $\frac{2\pi}{3}$ donne de bons résultats.

1. Pour être précis, représenter la déformation de la texture initiale, par exemple avec une grille, ferait appel aux « coordonnées texture inverses » : vers quel endroit de l'écran un point donné de la texture s'est-il déplacé ? Pour le calcul de l'advection, on a besoin de connaître les coordonnées textures « usuelles » : de quel endroit de la texture provient tel point de l'écran ?

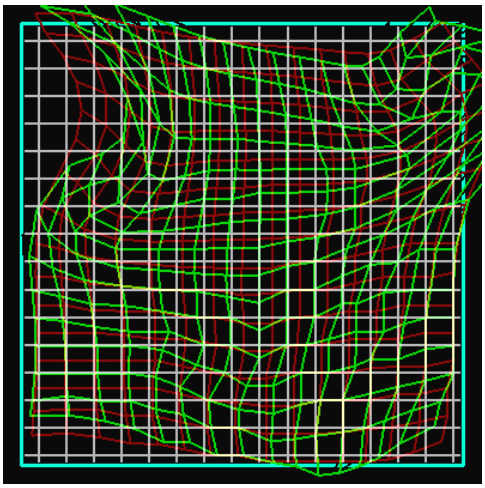

FIG. 6 – Grille déformée par le mouvement du fluide

Le paramètre λ, appelé *latence*, contrôle la durée de vie des textures. Son choix est critique :

- trop faible, la texture semble sautiller sans vraiment avancer : l'œil identifie bien chaque élément de bruit lorsqu'il est régénéré, et il n'y a pas d'illusion de mouvement, mais l'impression d'un retour en arrière ;
- trop élevée, la déformation de la texture devient excessive, ce qui la dénature.

1.3.2 Latence adaptative

Cependant, la vitesse moyenne et le taux de déformation peuvent varier beaucoup d'un point à un autre du fluide. Une valeur de latence unique ne donnera pas de bons résultats pour l'ensemble du fluide.

Partant de ce constat, [Ney03] propose d'utiliser une latence adaptée localement à la déformation maximale constatée. Dans les zones de faible mouvement, la latence est élevée ; là où les déplacements sont plus importants, la latence est plus courte. Cela donne de bien meilleurs résultats globaux.

En pratique, il faut calculer l'advection trois fois avec des valeurs de latence différentes, puis choisir localement quelle couche utiliser en fonction de la déformation mesurée. Un point délicat est de réussir le raccordement entre les différentes couches sans créer trop de *ghosting* — nous expliquons ce problème au paragraphe suivant.

Nous n'utiliserons pas cette idée, parce qu'elle est assez lourde à mettre en œuvre : il faut quasiment tripler tous les calculs ! Nous conservons une latence globale unique, et nous le payons par quelques artefacts, en particulier dans les zones de forte déformation.

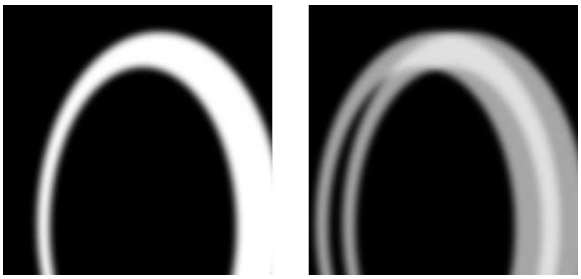

FIG. 7 – Exemple de *ghosting*

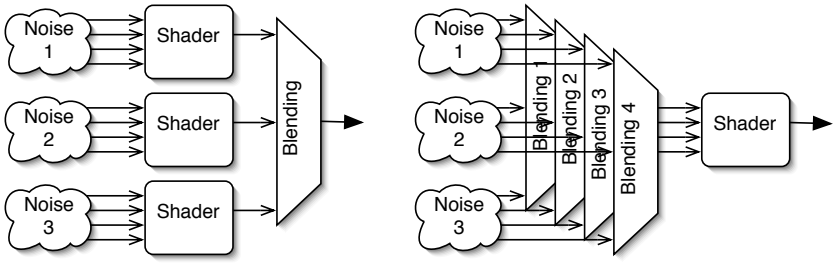

FIG. 8 – Mélange intuitif contre mélange intelligent

1.3.3 Mélange de textures procédurales

Le mécanisme de régénération conduit à superposer trois textures d'opacités variables, chacune contenant un bruit fractal. Leur mélange est susceptible de créer du *ghosting*.

Le *ghosting* est un artefact qui se produit lorsqu'on superpose des textures d'une opacité strictement inférieure à 1 (*i.e.* partiellement transparentes), et que l'œil continue de distinguer les motifs provenant des différentes textures de départ. Le phénomène est encore plus flagrant lorsqu'il entraîne le dédoublement d'un élément, comme illustré par la figure 7. Il s'accompagne d'une perte de constraste.

Dans le cas d'un bruit procédural, on peut améliorer cet état des choses. En effet, les différentes échelles de bruit sont combinées selon un calcul assez complexe, qui peut ne pas être linéaire — il est courant d'utiliser de valeurs absolues. Il semblerait intuitif de calculer chaque couche de bruit, puis de mélanger les résultats. Cependant, l'expérience prouve qu'on obtient de bien meilleurs résultats en mélangeant les bruits obtenus échelle par échelle, puis en combinant le résultat, comme le montre le schéma 8. De surcroît, on économise des calculs, car la combinaison des échelles est souvent plus compliquée qu'une somme pondérée.

Cette méthode n'élimine pas totalement le *ghosting*, mais elle le réduit

considérablement. En effet, il se rencontre surtout lorsque l'on mélange des textures dont les motifs sont suffisamment marqués pour rester identifiables. Or, un simple bruit de Perlin n'offre pas de structures marquées, contrairement à un bruit final traité. Le mélange de plusieurs couches de bruit de Perlin à la même échelle pourra donc induire une certaine perte de contraste, qu'il faudra compenser dans la suite des calculs, mais pas vraiment de *ghosting*. En fonction de la combinaison des différentes échelles qui est ensuite utilisée, le *ghosting* pourrait se révéler, mais l'expérience montre que ça n'est généralement pas le cas.

1.3.4 Animation des petites échelles

Comme expliqué en introduction, advecter passivement une texture figée est insuffisant. Le flow-noise est donc utilisé pour animer les petites échelles. L'article soulève le problème de l'ajustement de la vitesse de rotation des gradients, non seulement en fonction de l'échelle, mais aussi en fonction de la position spatiale. L'idée proposée est d'exploiter des résultats physiques, et en particulier la théorie de Kolmogorov, exprimée en vorticité, pour déterminer cette vitesse de rotation.

Kolmogorov affirme que dans un certain domaine spectral, on observe une *cascade d'énergie* dans le domaine de Fourier. L'énergie est introduite par des mouvements à grande échelle, puis se propage vers les petites échelles avant de se transformer en agitation moléculaire. Le mécanisme de cette cascade d'énergie se comprend intuitivement : les grands mouvements dans un fluide abandonné à lui-même se fractionnement en des tourbillons de plus en plus petits et de plus en plus rapides. L'équation qui décrit ce phénomène peut aussi s'exprimer en termes de la vorticité du fluide, qui est définie comme le rotationnel de son champ de vitesse. Cette grandeur nous intéresse particulièrement, car elle contrôle les tourbillons dans le fluide.

[Ney03] reproduit ce mécanisme en conservant une mesure de la vorticité en chaque point de la grille de discrétisation, et pour chaque échelle utilisée par le flow-noise. Une certaine quantité d'énergie, exprimée par une valeur de vorticité, est injectée à la plus grande échelle à partir des données du simulateur de fluides, puis elle est propagée vers les petites échelles, à un taux contrôlé par des constantes de temps. La vitesse de rotation des générateurs du flow-noise est alors déterminée localement pour chaque échelle à partir de la vorticité calculée.

Ceci reproduit qualitativement le mécanisme de la cascade de Kolmogorov. Le réalisme physique dépend essentiellement d'un choix judicieux des constantes de temps.

Nous retenons l'idée de la cascade de Kolmogorov, même si nous l'exploiterons plus simplement pour gagner en vitesse. Nous proposons un calcul énergétique détaillé à la partie 2.2.

2 Animation améliorée du bruit de Perlin

Nous décrivons ici les deux principales améliorations que nous apportons aux algorithmes précédents : la réalisation d'un flow-noise sans enroulements et une analyse énergétique basée sur la théorie de la turbulence de Kolmogorov.

2.1 Flow-noise optimisé

2.1.1 Flow-noise simple en 2D

Pour notre bruit de Perlin animé, nous allons reprendre l'idée des gradients rotatifs exposée en 1.2.1. Or, dans le plan, les gradients peuvent s'exprimer en coordonnées polaires, par un angle et une norme. Suivant l'idée du bruit de Perlin amélioré, nous pensons qu'il n'est pas nécessaire d'avoir des vecteurs complètement aléatoires : nous décidons donc de fixer la norme des gradients à 1. Nos gradients sont alors définis par un seul paramètre, leur angle de rotation, qui est de surcroît celui sur lequel nous voulons jouer pour créer l'animation !

Bien sûr, cette simplification augmente la régularité du bruit. Mais elle a plusieurs intérêts :

- n'avoir qu'un paramètre pour contrôler le bruit allège l'implémentation ;
- les ondelettes qui génèrent le bruit sont toutes isomorphes par rotation autour de l'axe vertical à celle représentée sur la figure 9 : à vitesse de rotation identique, elles simulent la même quantité d'énergie, ce qui facilite l'analyse physique.

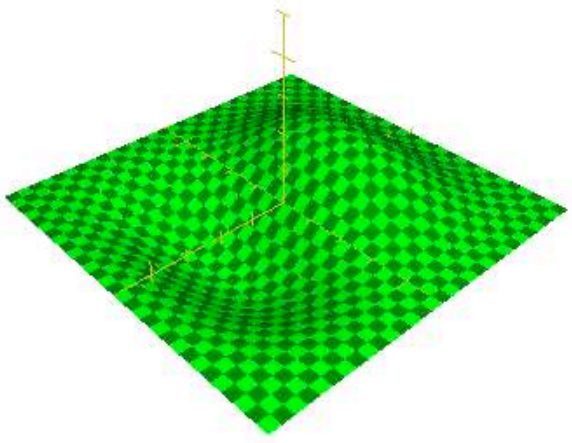

FIG. 9 – Générateur de bruit de Perlin

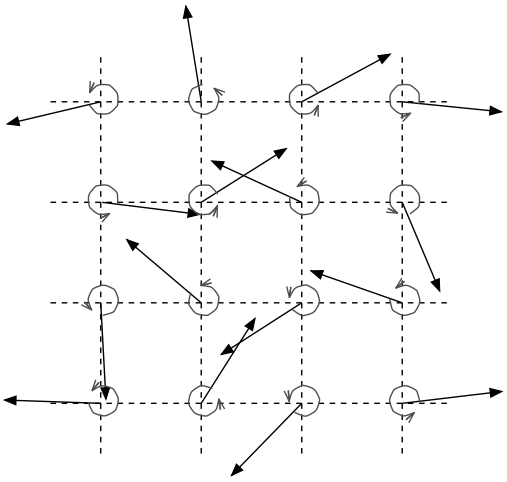

FIG. 10 – Gradients rotatifs

En cas de besoin, on pourrait ré-introduire la norme des gradients. Mais il faudrait alors décider comment la faire varier. Si cette norme est constante, les endroits où elle est faible sembleront toujours moins agités que là où elle est élevée, parce que l'ondelette associée ne transportera pas la même quantité d'énergie. Soit on considère que cela se produira à suffisamment petite échelle pour être statistiquement constant ; soit on décide de corréler norme et vitesse de rotation du gradient pour conserver l'énergie des ondelettes constante.

2.1.2 Flow-noise sans enroulements

Le problème des enroulements Dans son implémentation des textures advectées [Ney03], Fabrice Neyret s'est heurté à un problème d'« enroulements » qui apparaissent dans le flow-noise. Voici l'explication qu'il en donne : imaginons une déformation continue d'une texture plane, qui ait localement une rotation nulle autour d'un point A, et de dix tours autour du point B. Il est évident que la texture s'enroule autour de B, et qu'en rencontre les bras d'une spirale quand on se déplace de A vers B. Cette déstructuration de la texture était heureusement masquée par la latence adaptative, parce que la texture était régénérée rapidement dans les zones de fortes déformation où les enroulements apparaissaient.

Cette solution ne nous satisfait pas, parce que ces enroulements n'ont aucune raison d'être, en vertu de la remarque faite en 1.2 : *à chaque instant, le bruit conserve sa structure* — et pourtant, ils sont également apparus spontanément dans notre première implémentation ! De plus, nous voulons éviter la latence adaptative qui est lourde à gérer. Réaliser un flow-noise

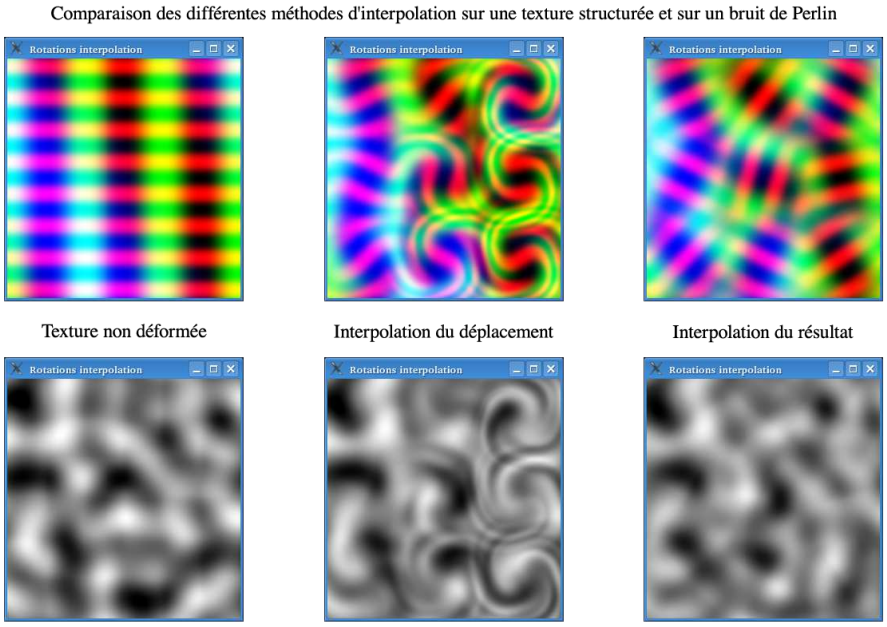

FIG. 11 – Tests d'interpolation d'un champ de rotations

sans enroulements était donc un objectif important.

Interpolation d'un champ de rotations Nous faisons une petite digression pour expliquer les enroulements qui peuvent apparaître lorsque l'on cherche à ajouter des rotations à une texture.

Considérons la problématique générale de la déformation d'une texture de telle sorte qu'elle respecte localement un champ de rotations défini sur une grille. Dans notre cas, ce champ de rotations permet de contrôler l'agitation du fluide. Entre les points de la grille, si l'on interpole les rotations ou le déplacement, on provoque des enroulements. Ce phénomène est inévitable dès que l'on déforme continûment le plan en respectant le champ de rotations.

Pour résoudre cette problématique, il faut abandonner l'idée d'une transformation géométrique continue du plan. On considère alors que chaque point à coordonnées entières a une « zone d'influence », qu'il entraîne dans sa rotation à vitesse angulaire uniforme.

La figure 11 montre le résultats de nos tests sur l'interpolation de rotations. Nous considérons deux textures fixes, une avec une géométrie bien marquée pour mieux visualiser les transformations effectuées, et un bruit de Perlin. Nous montrons à chaque fois successivement :

– la texture de départ,

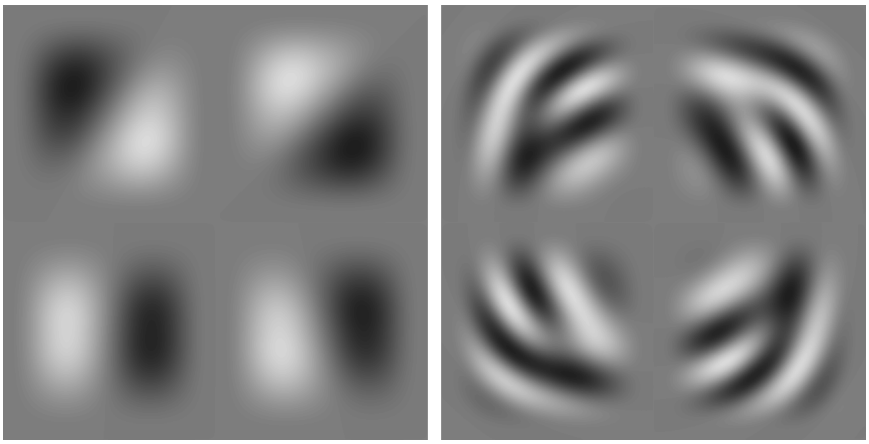

FIG. 12 – Ondelettes non déformées, et victimes d'enroulements

– la déformée de la texture par une transformation spatialement continue du plan, que nous nommons *interpolation de déplacement*,
– la texture obtenue par notre méthode, l'*interpolation du résultat*.

On constate que les enroulements, même faibles, ne sont pas acceptables sur une texture de bruit. Par contre, ne pas utiliser une transformation géométrique continue n'est pas choquant, parce que le résultat reste continu. On voit que le contraste du bruit est diminué ; c'est un artefact lié au fait que le pas des rotations ne correspond pas à l'échelle du bruit et qui disparaît lorsqu'on lie les rotations aux générateurs du bruit.

Comme notre flow-noise intègre les rotations dans la structure même du bruit de Perlin, cette complexité était complètement masquée.

Élimination des enroulements Les enroulements constatés provenaient du fait que la rotation à ajouter au générateur était calculée au point dessiné, et non au centre du générateur. Cela revenait à assurer la continuité géométrique de la transformation. Du coup, à un instant donnée, la rotation n'était pas uniforme que la surface du générateur, déformant ce dernier et provoquant les enroulements. Le problème se révélait donc dès que la vitesse de rotation n'était pas uniforme. La figure 12, où les ondelettes ont été écartées les unes de autres, isole le problème et montre les enroulements créés par un champ de rotation en $e^{1-r^2/2}$.

Pour corriger le problème, il suffit d'implémenter avec précision l'algorithme, c'est à dire en prenant pour chaque générateur la valeur de rotation en son centre. Cela impacte négativement les performances : au lieu d'une valeur par couche, il en faut quatre, puisque quatre générateurs se superposent en chaque point du plan. On perd aussi un peu de l'esprit procédural : les informations *au point courant* ne suffisent plus à calculer la texture.

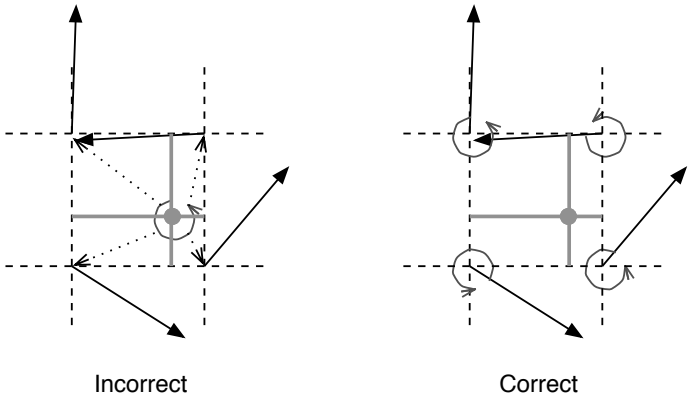

FIG. 13 – Valeur de rotation au point vs. valeur de rotation aux sommets

2.1.3 Animation des gradients

Nous aurons une information sur la vitesse de rotation des gradients grâce aux résultats de la partie 2.2. Se pose la question du sens de la rotation.

On pense spontanément à orienter la rotation en fonction de l'orientation de la vorticité. Mais si l'on anime tous les gradients par une rotation dans le même sens — et ce sera le cas localement, le mouvement semble peu naturel. En effet, au milieu des arêtes de la grille, les mouvements des deux générateurs les plus proches vont en sens contraires. Par ailleurs, cette intuition n'a pas de base physique : la turbulence est essentiellement un phénomène désorganisé.

Alterner le sens de rotation des générateurs selon un damier diminue des discontinuités du champ de vitesses perçu, et donne une animation visuellement plus agréable. Faute d'une théorie plus précise, nous nous en tiendrons à ce choix simple et visuellement satisfaisant.

2.1.4 Passage en 3D

Ce n'est pas trivial, parce qu'il faut également donner des axes de rotation aux générateurs. On pourrait en première approche les choisir aléatoires et fixes, même si c'est irréaliste ; la question de leur animation est un problème ouvert.

2.2 Contrôle spectral de la turbulence

2.2.1 Contexte

Ayant défini la manière dont nous calculons le flow-noise, nous revenons à notre problématique : enrichir l'animation des petites échelles d'un fluide.

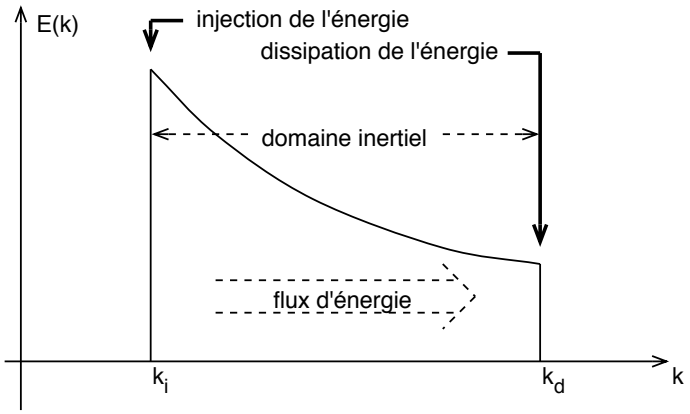

FIG. 14 – Cascade d'énergie de Kologorov

On suppose qu'un autre mécanisme calcule les grandes échelles et fournit une mesure de la vorticité à l'échelle de la grille en chaque point de celle-ci.

Dans cette partie, nous nous plaçons dans un contexte tridimensionnel, car la turbulence physique n'existe pas en deux dimensions. Nous travaillons sur un grille G_0 de pas λ ; en ses sommets se trouvent des ondelettes de bruit de taille 2λ tournant uniformément à la vitesse $\dot{\theta}_0$. Nous ajoutons ensuite pour j entier une grille G_j homothétique à G_0 dans un rapport 2^{-j}. G_j a un pas de $\lambda 2^{-j}$ et des ondelettes qui tournent à la vitesse $\dot{\theta}_j$. Nous cherchons à déterminer des valeurs plausibles pour les $\dot{\theta}_j$ en fonction de $\dot{\theta}_0$.

Nous appliquerons directement les résultats obtenus en 3D à notre simulation en 2D. Il y a là un raccourci, mais on ne sait pas à quoi est censé correspondre le résultat de notre flow-noise : une texture en coupe ? une déformation ? une intégrale d'une densité ? un paramètre de surface ? Dans le doute, faute d'interprétation univoque en 2D, nous faisons au plus simple.

2.2.2 Turbulence et cascade de Kolmogorov

Pour une approche classique de la théorie de la vorticité et de la turbulence dans un fluide, nous renvoyons à [Cho94]. Une approche plus pratique, appliquée aux fluides géophysiques se trouve dans [FLF]. Nous nous intéressons en particulier à l'approche énergétique qui y est proposée. Elle aboutit à l'équation définissant le spectre de Kolmogorov (1941) :

$$E(k) = C_K \epsilon^{2/3} k^{-5/3} \tag{1}$$

où $k = 2\pi/\lambda$ est le nombre d'onde, C_K la constante de Kolmogorov, ϵ le flux spectral d'énergie, et $E(k)$ la densité spectrale d'énergie cinétique, autrement dit l'énergie cinétique au nombre d'onde k.

Cette équation est valable dans un domaine spectral $k_i < k < k_d$, appelé domaine inertiel. k_i est le nombre d'onde où l'énergie est injectée par les mouvements de grande amplitude du fluide, et k_d celui où elle est absorbée par la viscosité. L'énergie est transférée vers les nombres d'ondes croissants (*i.e.* les petites échelles) au taux ϵ, supposé spatialement constant. Ce transfert est du à des interactions non-linéaires. La théorie de Kolmogorov affirme que les seuls paramètres importants dans le domaine inertiel sont le flux d'énergie ϵ et le nombre d'onde k, et non la viscosité ou les contraintes, pas exemple.

Notre flow-noise va ajouter des détails au mouvement du fluide à des échelles approximativement comprises entre le pas de la grille de simulation, et la taille du pixel. Il faut s'assurer que ce domaine est inclus dans le domaine inertiel. Notons déjà qu'on a bien $k_i \ll k_d$ quand le nombre de Reynolds du fluide est grand. Notre pré-requis est réaliste pour k_d car λ_d est généralement de l'ordre de grandeur de l'échelle moléculaire, donc très petit devant la taille d'un pixel. Selon l'écoulement considéré, il peut être plus ou moins réaliste du côté de k_i, la cascade d'énergie se faisant plutôt à de petites échelles.

2.2.3 Analyse énergétique

Dans notre modèle, nous allons supposer que l'énergie dans la bande spectrale $[k_j, k_{j+1}](k_{j+1} = 2k_j)$ est entièrement apportée par les générateurs de taille $\lambda_j = 2\pi/k_j$. Cette hypothèse paraît raisonnable, éventuellement à un facteur d'échelle près.

On note $g_j(\vec{r})$ l'ondelette à l'échelle j : $g_j(\vec{r}) = g(\lambda_j^{-1}\vec{r})$, W_j le poids de cette échelle et on suppose la densité du fluide incompressible normalisée à 1. On suppose l'ondelette à symétrie sphérique : $g(\vec{r}) = g(r)$, afin que son l'énergie ne dépende pas de l'orientation de son vecteur rotation.

L'énergie apportée par une ondelette se calcule par :

$$
\begin{aligned}
E_1(j) &= W_j \int_{\mathbb{R}^3} g_j(\vec{r}) \frac{1}{2} \left(\vec{\dot{\theta}_j} \wedge \vec{r} \right)^2 \mathrm{d}\vec{r} \\
E_1(j) &= W_j \left(\frac{k_j}{k_0} \right)^{-5} \int_{\mathbb{R}^3} g\left(\frac{k_j}{k_0} \vec{r} \right) \frac{1}{2} \left(\vec{\dot{\theta}_j} \wedge \frac{k_j}{k_0} \vec{r} \right)^2 \mathrm{d} \left(\frac{k_j}{k_0} \right)^3 \vec{r} \\
E_1(j) &= W_j \left(\frac{k_j}{k_0} \right)^{-5} {\dot{\theta}_j}^2 K_g
\end{aligned}
$$

où K_g est une constante dépendant de la forme de l'ondelette.

Dans le cas usuel, les générateurs occupent les sommets d'une grille 3D de pas $\lambda_j = 2^{-j}\lambda_0$ et la densité d'énergie dans la bande spectrale $[k_j, k_{j+1}]$ est alors :

$$
E_{[k_j, k_{j+1}]} = \left(\frac{k_j}{k_0} \right)^{-2} {\dot{\theta}_j}^2 K_g = 2^{-2j} {\dot{\theta}_j}^2 K_g
$$

Maintenant, nous pouvons évaluer cette même énergie d'une deuxième manière grâce à l'équation de Kolmogorov :

$$\begin{aligned} E_{[k_j,k_{j+1}]} &= \int_{k_j}^{k_{j+1}} E(k)\,\mathrm{d}k \\ &= \int_{k_j}^{k_{j+1}} C_K \epsilon^{2/3} k^{-5/3}\,\mathrm{d}k \\ &= C_K \epsilon^{2/3} \int_{k_j}^{2k_j} k^{-5/3}\,\mathrm{d}k \\ E_{[k_j,k_{j+1}]} &= C_K \epsilon^{2/3} \frac{3}{2}\left(1 - 2^{-2/3}\right) k_j^{-2/3} \\ E_{[k_j,k_{j+1}]} &\approx 0.555 C_K \epsilon^{2/3} k_j^{-2/3} \end{aligned}$$

On en déduit que :

$$W_j \left(\frac{k_j}{k_0}\right)^{-2} \dot{\theta_j}^2 K_g \approx 0.555 C_K \epsilon^{2/3} k_j^{-2/3}$$

et, après élimination de toutes les constantes :

$$W_j \dot{\theta_j}^2 \sim k_j^{4/3}$$

Le cœfficient de proportionnalité dans cette dernière relation dépend de la forme de l'ondelette via K_G et de l'énergie injectée à grande échelle via ϵ. Nous ne calculons pas explicitement ce cœfficient. Nous préférons estimer la vitesse de rotation locale à l'échelle de la grille de discrétisation à partir du champ de vitesse ; nous connaissons alors W_0, $\dot{\theta_0}$ et k_0. Ensuite, la relation de proportionnalité nous suffit pour calculer les échelles inférieures.

2.2.4 Un nouveau modèle pour l'animation des petites échelles

Un modèle simple utiliserait typiquement les paramètres suivants, qui donnent des résultat acceptables : $k_j = 2^j k_0$ (i.e. $\lambda_j = 2^{-j}\lambda_0$), $\dot{\theta_j} = 2^j \dot{\theta_0}$ et $W_j = 2^{-j} W_0$.

Il paraît judicieux de conserver $\lambda_j = 2^{-j}\lambda_0$ par commodité d'implémenation. Reste donc à définir des valeurs pour $\dot{\theta_j}$ et W_j telles que :

$$W_j \dot{\theta_j}^2 \sim \left(2^j\right)^{4/3} \tag{2}$$

En supposant que nous gardons $\dot{\theta_j} = 2^j \dot{\theta_0}$, le tableau 15 montre la différence entre des valeurs de W_j/W_0 par défaut (en 2^{-j}), et celles résultant de notre calcul (en $2^{-2j/3}$).

Cela montre que l'importance des petites échelles est facilement sous estimée. Au bout de quatre échelles fractales, une dimension courante, l'erreur va du simple au double !

θ_j/θ_0	1	2	4	8	16	32
W_j/W_0 (par défaut)	1.0	0.50	0.250	0.125	0.0625	0.03125
W_j/W_0 (correct)	1.0	0.63	0.397	0.250	0.1575	0.09921

FIG. 15 – Comparaison de poids des différentes échelles

2.2.5 Comparaison avec les modèles précédents

[Ney03] proposait de reproduire directement la cascade d'énergie par un mécanisme de flux d'énergie entre les échelles, contrôlé par des constantes de temps. Cette manière de contrôler l'activité du fluide nous paraît moins intuitive en terme de résultats. Elle présente également le défaut d'être plus lourde en calculs puisqu'il faut stocker les valeurs de vorticité à chaque échelle en chaque point de la grille et les faire évoluer, c'est à dire calculer dynamiquement les θ_j à chaque échelle.

Notre méthode nécessite seulement de calculer θ_0 en chaque point ; les θ_j lui sont proportionnels et les W_j sont fixes. De plus, les coefficients à fixer (facteurs de poids et de vitesse de rotation des ondelettes) ont une influence intuitive sur le résultat : on définit le contraste de chaque échelle, et son taux d'agitation. En revanche, nous perdons l'aspect dynamique lié au temps de transfert de l'énergie : les petites échelles sont directement influencées par les grandes échelles, ce qui veut dire que notre solution est plutôt adaptée à des régimes continus. Cette restriction pourrait rendre certains effets, comme une atténuation rapide de l'agitation, irréalistes. Il est difficile de dire si l'on voit vraiment une différence en pratique ; il faudrait étudier les temps de transfert caractéristiques dans les fluides usuels. S'ils restent de l'ordre de l'intervalle de temps entre deux images successives, le défaut ne se verra pas.

3 Implémentation sur carte graphique

Nous avons réalisé une première implémentation de notre algorithme sur CPU. Le processeur devait à la fois effectuer les calculs coûteux du simulateur de fluides, et le rendu de la texture ; il en était incapable en temps réel dès que la résolution dépassait 128×128.

Nous avons donc décidé de calculer le flow-noise sur la carte graphique, profitant de son énorme puissance de calcul flottant et de sa capacité de calcul parallèle, même si son jeu d'instructions réduit est un handicap pour un algorithme aussi complexe. Cette réalisation a plusieurs intérêts : elle permet de tester interactivement l'algorithme à moyenne résolution, ce qui aide à l'améliorer ; elle ouvre aussi la voie à une utilisation de cet algorithme dans les jeux vidéos.

Notre maquette logicielle est écrite en C++. Elle utilise Qt [Qt] et la libQGLViewer [LQV] pour l'interface. Elle s'appuie sur la LibSL[Lef] pour les structures de données complexes et l'interface avec la carte graphique, en particulier le chargement des textures et du shader. La partie graphique repose sur OpenGL [OGL] et sur Cg [Cg, FK03] pour le *pixel shader*.

Le schéma 16 décrit le flux de données qui permet le calcul de la texture. Nous expliquerons quelles données doivent être calculées en amont du rendu, avant de détailler quelques points particuliers du *pixel shader*.

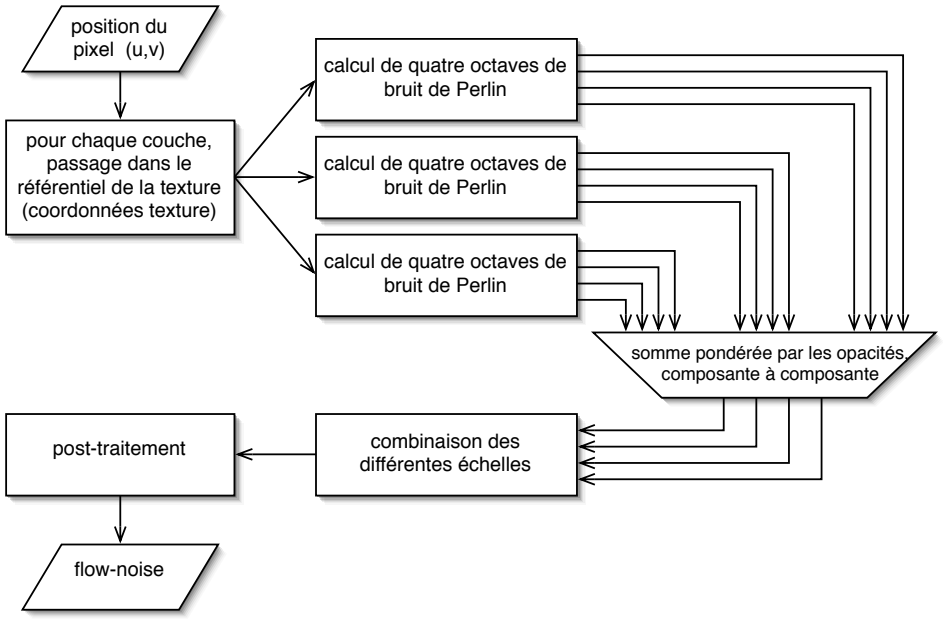

FIG. 16 – Schéma de l'algorithme de rendu

3.1 Pré-requis

3.1.1 Simulateur de fluides

Nous devons avoir un fluide comme base sur laquelle appliquer nos textures ! Nous utilisons le simulateur de fluides 2D décrit par Jos Stam dans [Sta03]. Son objectif était la simulation dynamique de fluides en temps réel pour des jeux vidéo, ce qui est tout à fait complémentaire avec notre projet. Son algorithme est inconditionnellement stable, au prix d'une précision médiocre ; nous pouvons utiliser des pas de temps aussi grands que nous le voulons. Nous obtenons à chaque instant un champ de vitesses, et nous pouvons interagir en temps réel avec le fluide. Nous pouvons aussi advecter des champs scalaires avec le fluide.

Le code de départ étant particulièrement simple, nous avons pu lui apporter des modifications importantes :

- Nous avons amélioré la manière dont l'utilisateur contrôle le fluide.
- Nous avons traduit toutes les appels à des tableaux pour utiliser les structures de la LibSL, qui permettent de transmettre facilement les textures à la carte graphique.
- Nous avons codé des conditions aux limites variées pour obtenir des styles de mouvement différents.
- Nous avons écrit une fonction qui mesure localement la vorticité, ce dont notre algorithme a besoin.
- Nous avons ajouté un *vorticity confinement* [SU94] rudimentaire. Il s'agit d'amplifier localement la turbulence du fluide ; sans cela, les tourbillons sont absorbés extrêmement rapidement par la dissipation numérique, et cela complique les tests. Nous mesurons simplement le rotationnel du champ de vitesse, et nous l'augmentons d'un certain facteur, en modifiant localement le champ de vitesse à somme nulle. Sans surprise, forcer sur le *vorticity confinment* rend le simulateur instable !
- Nous avons ajouté à la fonction d'advection déjà présente une fonction d'« advection inverse » pour maintenir les coordonnées textures inverses.
- Nous avons défini une fonction pour calculer les valeurs de rotation dans le référentiel d'un texture, car il est beaucoup plus efficace de pré-calculer cette transformation.

L'utilisateur peut ajouter du mouvement et de la densité dans le fluide à la souris. Le simulateur prend ensuite en charge le calcul du champ de vitesse, et de toutes les autres données nécessaires au calcul du flow-noise.

3.1.2 Données d'entrée

Notre algorithme prend en entrée les données suivantes, qui doivent être calculées sur CPU et envoyées à la carte graphique (voir également en A.3.1) :

- un champ de densité, auquel nous allons ajouter du bruit,
- pour chacune des trois couches de textures :
  - son opacité,
  - le tableau de ses coordonnées textures — lui-même passé sous forme de texture,
  - un tableau de valeurs de rotation, exprimées dans le référentiel de coordonnées de la texture,
- un tableau de valeurs aléatoires pour effectuer la décorrélation nécessaire au calcul du bruit de Perlin,
- plusieurs scalaires qui contrôlent les paramètres spatio-temporels du flow-noise, et les transformations appliquées au bruit obtenu.

Bien comprendre ce qui est calculé dans le référentiel du fluide et ce qui l'est dans le référentiel de la couche de texture est une de plus grosses difficultés conceptuelles de l'algorithme. Par exemple, les valeurs de rotation doivent être exprimées dans le référentiel de la texture, parce qu'on en a besoin lorsqu'on calcule le bruit de Perlin, ce qui impose d'utiliser une grille liée à la texture. Le calcul de ces valeurs requiert donc de connaître les coordonnées textures inverses, pour effectuer le changement de référentiel sur les valeurs de rotation fournies dans le référentiel du fluide par la simulation. Ceci force à maintenir à jour des coordonnées textures inverses en plus des coordonnées textures directes.

3.1.3 Interface

L'interface du programme est reproduite à la figure 17. On y trouve :
- les propriétés du fluide en haut à gauche,
- les divers modes d'affichage en bas à gauche,
- les contrôles généraux du shader au milieu,
- les réglages des propriétés spatio-temporelles du bruit dans la colonne de droite,
- et bien sûr la visualisation principale.

3.2 Algorithme

Nous allons commenter quelques étapes du fonctionnement de notre algorithme (figure 16). Nous supposons que le simulateur de fluides a actualisé et transmis à la carte graphique toutes les données mentionnées en 3.1.2. Le code source complet et commenté du *pixel shader* est fourni à l'annexe A.1.

3.2.1 Calcul des coordonnées texture

Cette opération doit se faire une fois dans chaque couche, en fonction des données d'advection de cette couche. Il faut interpoler les valeurs fournies par le simulateur de fluides sur sa grille grossière. Ces valeurs étant passées

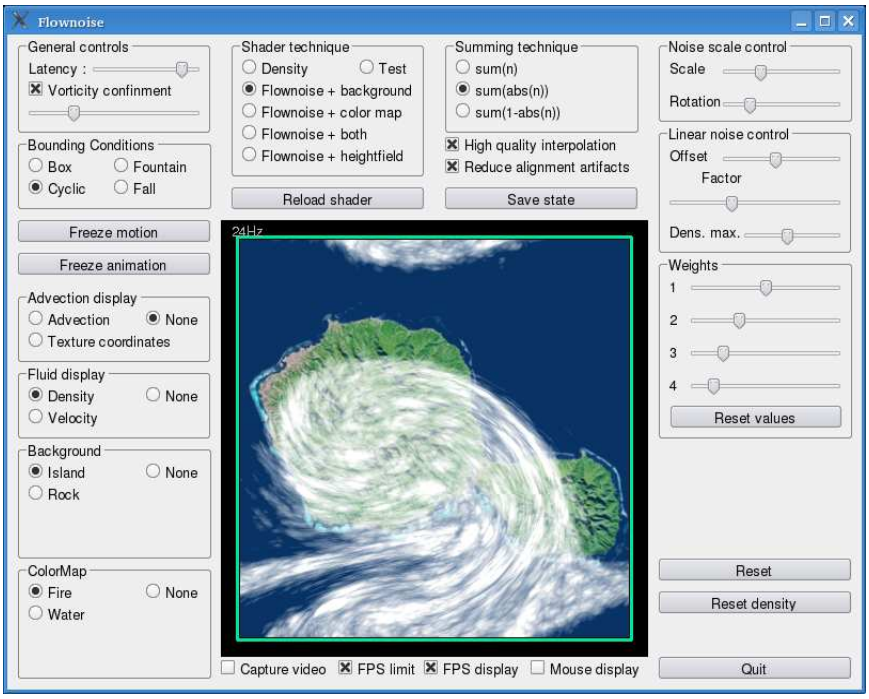

Fig. 17 – Interface du programme de test

dans une texture 16 bits — avec l'abscisse et l'ordonnée dans les deux premières composantes, on est tenté d'utiliser l'interpolation matérielle dans cette texture.

Cependant, dès que l'on agrandit la texture, on voit des « blocs » comme sur l'image de gauche de la figure 18. Ce phénomène s'observe même à des facteurs de grossissement modérés, de l'ordre de $\times 10$, et il provoque en plus une sorte de « sautillement » désagréable de la texture.

Ces artefacts proviennent de ce que l'interpolateur matériel ne fonctionne qu'en 16 bits, et que les flottants 16 bits des cartes graphiques sont très limités. En l'occurrence, la taille des blocs est presque certainement $1/2^{10}$, c'est à dire la précision de la mantisse de ces nombres. Au sein de chacun des blocs, le shader considère que tous les points ont la même position dans la couche de texture.

Pour dépasser cette limitation, nous avons choisi d'effectuer l'interpolation manuellement. Cela nous coûte la décomposition de la position (u, v) dans la grille en partie entière et décimale, quatre accès texture au lieu d'un pour chaque couche et l'interpolation proprement dite. Cependant, le surcoût est moins considérable qu'on ne pourrait l'imaginer, pour plusieurs raisons :

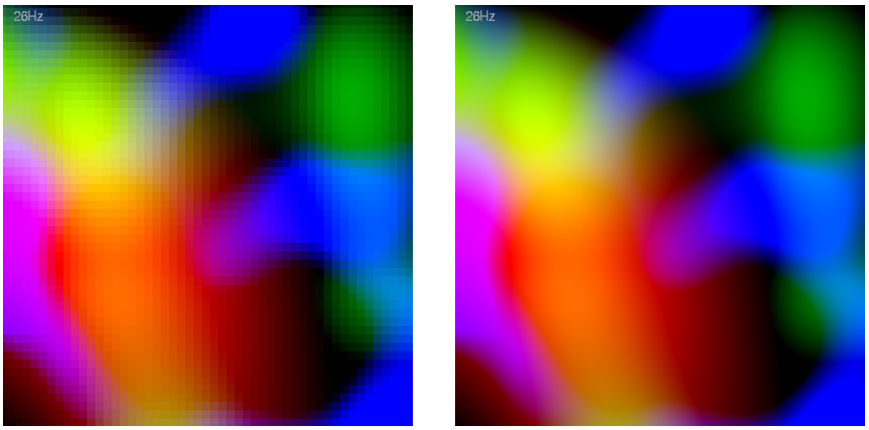

FIG. 18 – Interpolation matérielle 16 bits vs. interpolation logicielle 32 bits

– on désactive l'interpolation dans la texture, ce qui accélère l'accès,
– on n'effectue l'interpolation que sur les deux coordonnées qui nous sont nécessaires,
– on profite du cache mémoire en accédant à quatre positions contiguës,
– l'interpolation est très optimisée : une instruction lui est dédiée.

En pratique, la différence de performances est imperceptible. Le choix de la méthode retenue est contrôlé par une option de compilation du shader. Notons que cela résout le problème des blocs, mais pas celui du « sautillement » : les coordonnées textures fournies en entrée étant toujours représentées par des flottants 16 bits, la couche se déplace toujours par incréments de $1/2^{10}$, ce qui peut se remarquer lors de déplacements trop lents.

Nous retenons de cette difficulté que les flottants 16 bits sont parfaits pour transporter des informations de couleur, mais trop limités pour les informations de position. Nous avons donc modifié le shader en utilisant au maximum les flottants 32 bits pour toutes les données correspondant à des positions, dans les limites du nombre de registres flottants disponibles.

3.2.2 Calcul du bruit de Perlin

Le calcul du bruit de Perlin nécessite les coordonnées texture qu'on vient de calculer, un cœfficient de mise à l'échelle s et un coefficient de rotation r.

Les coordonnées textures sont décomposées en partie entière et décimale modulo s. La partie entière permet de récupérer les quatre valeurs de rotation pour les quatre générateurs voisins. Nous pré-traitons la texture qui contient les valeurs de décorrélation en plaçant les valeurs des sommets voisins dans les composantes y, z et w de la texture : ainsi les quatre valeurs de décorrélation sont récupérées en un seul accès texture. Enfin nous alternons

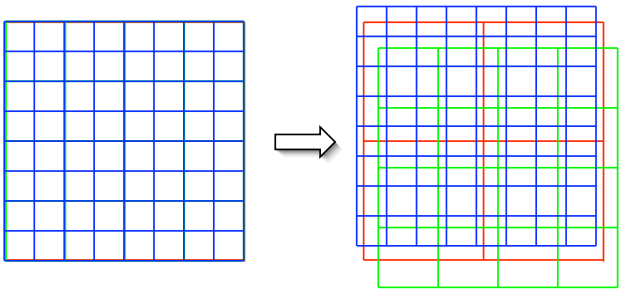

FIG. 19 – Décalages des grilles aux différentes échelles

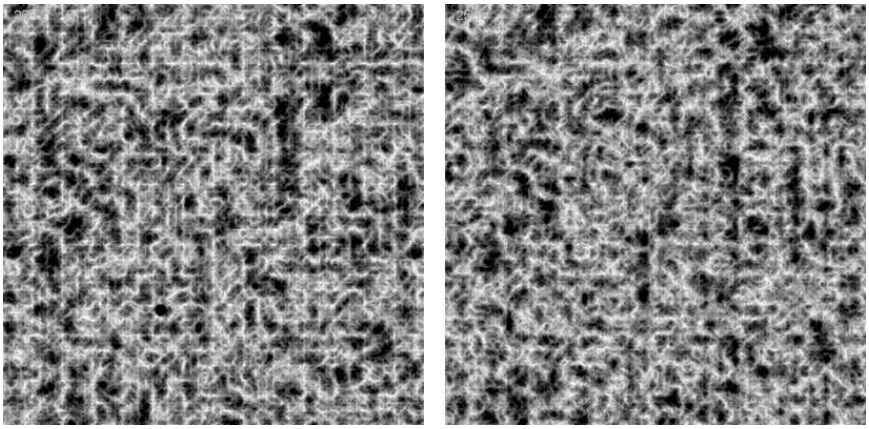

FIG. 20 – Réduction des artefacts par le décalage des grilles

les valeurs de rotation selon la parité de la position du point et nous les multiplions par r.

Il n'y a plus qu'à calculer les valeur des gradients au point considéré et à effectuer l'interpolation lissée, en utilisant la partie décimale des coordonnées, qui donne la position par rapport aux sommets voisins.

Nous avons rencontré des « artefacts d'alignement » : des lignes sombres ou claires, alignées sur les axes de la grille, et qui ont l'air très artificielles [2]. Pour diminuer ce problème, nous avons légèrement décalé les grilles aux différentes échelles (cf. figure 19), réduisant ainsi la régularité du bruit. Comme le montre la figure 20, ou la transformation du bruit a été choisie pour faire apparaître des lignes blanches, les résultats s'améliorent sensiblement avec le décalage des grilles. Notons toutefois que ce problème n'est flagrant que

2. Le phénomène est beaucoup plus évident sur écran que sur papier.

sur une texture immobile ; dès qu'elle commence à se déformer, on ne le remarque plus.

3.2.3 Mélange et post-traitement

Pour mélanger nos trois couches de texture, nous appliquons directement l'algorithme de mélange intelligent exposé en 1.3.3. La combinaison des différentes échelles utilise les fonctions classiques : somme, somme des valeurs absolues, un moins la somme des valeurs absolues. Nous définissons le poids des différentes échelles par défaut selon les résultats de 2.2.

Nous ajoutons quelques contrôles élémentaires sur le bruit :

- facteur de mise à l'échelle,
- facteur de rotation,
- transformation affine du résultat.

La combinaison du bruit et de la densité est une simple somme. Pour améliorer le résultat, nous proposons de majorer la densité avant d'effectuer la somme. On peut ainsi conserver du bruit dans le résultat là où la densité saturerait sinon.

3.3 Résultats

3.3.1 Performances obtenues

La vitesse d'exécution de notre programme est essentiellement limitée par le débit des *pixel shaders* de la carte graphique. Nous avons travaillé à une résolution de 400×400 pixels, sur une Quadro FX1400 : notre algorithme tournait alors légèrement au dessus de 25Hz, c'est à dire en temps réel.

Nous avons fait quelques statistiques (cf. A.2) sur le code compilé du shader. Celui-ci fait près de 800 instructions assembleur : on imagine le coût élevé de son exécution en chaque pixel. Elles montrent montre que l'essentiel du coût réside dans :

- Les 73 appels de texture, nécessaires pour remplacer les tableaux ; ils se décomposent comme suit :
  - 24 appels par couche : 4 pour obtenir la position spatiale précise — on peut réduire à 1 en utilisant l'interpolation matérielle comme expliqué en 3.2.1 — et ensuite pour chacune des 4 échelles, 4 appels pour obtenir la valeur de rotation des quatre générateurs voisins, et 1 appel pour la valeur de décorrélation : total $4 + 4 \times (4 + 1) = 24$.
  - 1 appel pour obtenir la densité.

  Nous ne sommes pas parvenus à réduire davantage ce coût sans créer d'artefacts.
- Les 48 cosinus et les 48 sinus : en effet, un point dépend de 48 ondelettes : 4 pour chacune des 4 échelles, pour chacune des 3 couches. L'ondelette étant définie par un angle, le calcul du cosinus et du sinus sont indispensables pour évaluer sa contribution

– Curieusement, le test de la parité des générateurs pour alterner leur sens de rotation représente un coût important ! La carte graphique est incapable de calculs entiers et elle ne sait pas évaluer simplement `alt = (i + j) % 2`. La solution la plus rapide est de calculer `alt = frac(dot((i,j),(0.5,0.5)))`, qui vaut alternativement 0 ou 0,5.

3.3.2 Quelques images

Il n'est pas très parlant de donner des images des résultats obtenus, puisque l'objectif de notre travail est justement l'animation du bruit ! On trouvera une vidéo de présentation à l'adresse suivante :
`http://www-evasion.imag.fr/Publications/2006/AN06/`

Voici quelques résultats obtenus :

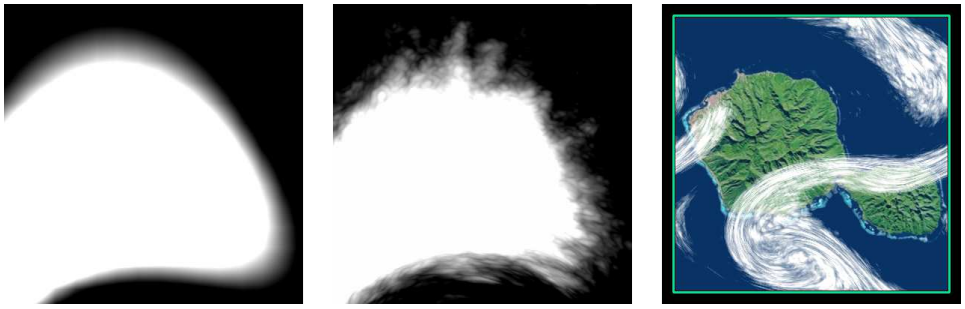

FIG. 21 – Densité, enrichissement par le flownoise et application à une couverture nuageuse

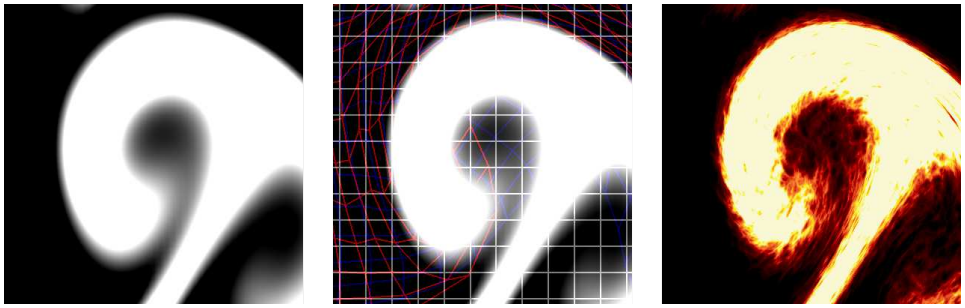

FIG. 22 – Densité, coordonnées texture et enrichissement avec une texture de flamme

Conclusion

Nous avons rempli l'essentiel des objectifs du stage : définir un modèle de flow-noise robuste et général, spécifier un champ de rotations amplifiant un champ de vitesses, et implémenter une maquette logicielle sur carte graphique. En revanche, privilégiant une orientation temps-réel, nous nous sommes contentés de flow-noise en 2D. Les travaux de Fabrice Neyret ont montré que le flow-noise volumique était possible ; aussi, nous pensons que notre algorithme fonctionnerait en 3D.

Nous obtenons un résultat temps réel à moyenne résolution, sans artefacts marqués, grâce à notre algorithme correct pour le flow-noise. Cependant, cela engendre un coût très élevé, en particulier pour l'accès aux valeurs de rotation. On peut imaginer plusieurs pistes pour améliorer les performances et les résultats.

Dans les jeux vidéos, les fluides sont souvent des éléments de décor avec lesquels le joueur ne peut pas interagir. On pourrait alors définir procéduralement les valeurs de rotation et éventuellement les coordonnées textures, ce qui supprimerait la plupart des accès texture et allégerait nettement le shader. Il deviendrait alors réaliste de l'utiliser pour des éléments de taille raisonnable : cascades, lacs de lave ou coulées de boue.

Une piste de recherche intéressante est l'exploitation de ce flow-noise pour la synthèse de nuages en temps réel, dans la lignée des travaux actuels d'Antoine Bouthors, Fabrice Neyret et Sylvain Lefebvre. Comme notre algorithme procédural est peu sensible à la résolution, on pourrait créer une couverture large avec un bon niveau de détails local, grâce à une approche par niveaux de détails (*e.g. LOD*). Le bruit fournirait une information de densité ou d'épaisseur du nuage. Comme les nuages se déforment lentement, on pourrait calculer des états du nuage à quelques secondes d'intervalle, et faire le rendu en interpolant entre les deux derniers états obtenus, en faisant attention au *ghosting*. On calculerait une partie de l'état du nuage à chaque frame, et on stockerait le résultat directement dans une texture. On obtiendrait alors une couverture nuageuse animée extrêmement détaillée pour un coût faible.

Remerciements

Je tiens à remercier Fabrice Neyret, mon tuteur, pour toutes ses explications ; Sylvain Lefebvre pour l'aide qu'il m'a apporté en matière de programmation sur carte graphique ; et toute l'équipe Evasion qui m'a accueilli pendant ce stage.

A Pixel shader — version simplifiée

Nous donnons ici le code source en langage Cg [Cg] de notre *pixel shader* de flow-noise, et nous le documentons. Nous ne présentons qu'une version minimale, qui ajoute du bruit à une densité. Il est ensuite facile d'ajouter un post-traitement sur le résultat, comme l'application d'une palette de couleurs. Le schéma de la figure 23 décrit l'exécution du shader.

A.1 Code source

```
1  // weights for blending results into a flownoise
2  uniform float3 timeWeights;     // opacity of each texture, varying over time
3  uniform float4 scaleWeights;    // weight of each scale
4
5  uniform float noiseOffset;      // linear control of noise
6  uniform float noiseFactor;
7  uniform float maximumDensity;  // limit density to prevent saturation
8
9
10 sampler2D decorrTex = sampler_state {
11   MinFilter = Nearest;
12   MagFilter = Nearest;
13 };
14 sampler2D densTex = sampler_state {
15   MinFilter = Linear;
16   MagFilter = Linear;
17 };
18 sampler2D rotTex = sampler_state {
19   MinFilter = Linear;
20   MagFilter = Linear;
21 };
22 #ifdef HQ_INTERPOLATION
23 sampler2D texCoords1Tex = sampler_state {
24   MinFilter = Nearest;
25   MagFilter = Nearest;
26 };
27 sampler2D texCoords2Tex = sampler_state {
28   MinFilter = Nearest;
29   MagFilter = Nearest;
30 };
31 sampler2D texCoords3Tex = sampler_state {
32   MinFilter = Nearest;
33   MagFilter = Nearest;
34 };
35 #else
36 sampler2D texCoords1Tex = sampler_state {
37   MinFilter = Linear;
38   MagFilter = Linear;
39 };
40 sampler2D texCoords2Tex = sampler_state {
41   MinFilter = Linear;
42   MagFilter = Linear;
43 };
44 sampler2D texCoords3Tex = sampler_state {
45   MinFilter = Linear;
46   MagFilter = Linear;
47 };
48 #endif
49
```

```
50 //// custom perlin noise
51
52 // sScale : space scale of the flownoise : size of a 'cell'
53 // rScale : rotation speed is multiplied by this value
54 half fp_perlin_noise_2D(float sScale, float rScale, float2 uv, int layer)
55 {
56 #ifdef HQ_INTERPOLATION
57   float2 c, ci, cf, f;
58 #else
59   half2 c, ci, cf, f;
60 #endif
61   half2 cs00, cs01, cs10, cs11;
62   half4 r;       // order : 00, 01, 10, 11
63   half a, b, alt;
64   // get int/float value of the point relatively to scale
65   c = uv/sScale; ci = floor(c); cf = c - ci;
66   alt = frac(dot(ci,float2(.5,.5))); // will be used for parity checking
67   // compute rotation values on the summits of the surrounding cube
68   // center of texture pixels is (0.5, 0.5)
69   r.x = tex2D(rotTex, (ci+float2(0.5,0.5))*sScale)[layer]; // corner 00 (lower left)
70   r.y = tex2D(rotTex, (ci+float2(0.5,1.5))*sScale)[layer]; // corner 01 (upper left)
71   r.z = tex2D(rotTex, (ci+float2(1.5,0.5))*sScale)[layer]; // corner 10 (lower left)
72   r.w = tex2D(rotTex, (ci+float2(1.5,1.5))*sScale)[layer]; // corner 11 (upper right)
73   // alternates rotation using the parity of ci.x + ci.y
74   // we check whether (ci.x + ci.y)/2 has a fractionnal part
75   if (alt) { r *= float4(1, -1, -1, 1); }
76       else { r *= float4(-1, 1, 1, -1); }
77   r += tex2D(decorrTex, (ci+0.5)*DECORR_TEXEL_SIZE);
78   r *= rScale;
79   // cos and sin of these values
80   sincos(r.x, cs00.y, cs00.x); sincos(r.y, cs01.y, cs01.x);
81   sincos(r.z, cs10.y, cs10.x); sincos(r.w, cs11.y, cs11.x);
82   // dot products + interpolation to compute Perlin Noise
83   f.x = smoothstep(0, 1, cf.x); f.y = smoothstep(0, 1, cf.y);
84   a = lerp(dot(float2(cf.x,cf.y),cs00), dot(float2(cf.x-1,cf.y),cs10), f.x);
85   b = lerp(dot(float2(cf.x,cf.y-1),cs01), dot(float2(cf.x-1,cf.y-1),cs11), f.x);
86   return lerp(a, b, f.y);
87 }
88
89 // this computes the flownoise, returning 4 noise values in [-1,1]
90 half4 fp_raw_flownoise_2D(float2 uv)
91 {
92   float4 n;
93   half4 nn;
94   half4 scaleCoeff = NOISE_SCALE*half4(1.0, 0.5, 0.25, 0.125);
95   half4 rotCoeff = ROTATION_FACTOR*half4(NOISE_ROTATION);
96   // interpolate texture coordinates
97 #ifdef HQ_INTERPOLATION
98   float2 tc;
99   // using 32 bits interpolation (instead of HW 16 bits interpolation)
100  // avoids some artifacts
101  float2 c, ci, cf;
102  float2 tc00, tc01, tc10, tc11;
103  c = uv/GRID_TEXEL_SIZE + 0.5; ci = floor(c); cf = c - ci;
104  tc00 = tex2D(texCoords1Tex, (ci+float2(-0.5,-0.5))*GRID_TEXEL_SIZE).xy;
105  tc01 = tex2D(texCoords1Tex, (ci+float2(-0.5, 0.5))*GRID_TEXEL_SIZE).xy;
106  tc10 = tex2D(texCoords1Tex, (ci+float2( 0.5,-0.5))*GRID_TEXEL_SIZE).xy;
107  tc11 = tex2D(texCoords1Tex, (ci+float2( 0.5, 0.5))*GRID_TEXEL_SIZE).xy;
108  tc = lerp(lerp(tc00, tc10, cf.x), lerp(tc01, tc11, cf.x), cf.y);
109 #define OFF1 float2( 0   , 0   )*OFFSET
110 #define OFF2 float2( 0.11, 0.04)*OFFSET
111 #define OFF3 float2(-0.07,-0.03)*OFFSET
```

```
112 #define OFF4 float2(-0.03, 0.01)*OFFSET
113 #else
114   half2 tc;
115   tc = tex2D(texCoords1Tex, uv).xy;
116 #define OFF1 half2( 0    , 0    )*OFFSET
117 #define OFF2 half2( 0.11, 0.04)*OFFSET
118 #define OFF3 half2(-0.07,-0.03)*OFFSET
119 #define OFF4 half2(-0.03, 0.01)*OFFSET
120 #endif
121   nn.x = fp_perlin_noise_2D(scaleCoeff.x, rotCoeff.x, tc+OFF1, 0);
122   nn.y = fp_perlin_noise_2D(scaleCoeff.y, rotCoeff.y, tc+OFF2, 0);
123   nn.z = fp_perlin_noise_2D(scaleCoeff.z, rotCoeff.z, tc+OFF3, 0);
124   nn.w = fp_perlin_noise_2D(scaleCoeff.w, rotCoeff.w, tc+OFF4, 0);
125   n = timeWeights.x*nn;
126
127 #ifdef HQ_INTERPOLATION
128   tc00 = tex2D(texCoords2Tex, (ci+float2(-0.5,-0.5))*GRID_TEXEL_SIZE).xy;
129   tc01 = tex2D(texCoords2Tex, (ci+float2(-0.5, 0.5))*GRID_TEXEL_SIZE).xy;
130   tc10 = tex2D(texCoords2Tex, (ci+float2( 0.5,-0.5))*GRID_TEXEL_SIZE).xy;
131   tc11 = tex2D(texCoords2Tex, (ci+float2( 0.5, 0.5))*GRID_TEXEL_SIZE).xy;
132   tc = lerp(lerp(tc00, tc10, cf.x), lerp(tc01, tc11, cf.x), cf.y);
133 #else
134   tc = tex2D(texCoords2Tex, uv).xy;
135 #endif
136   nn.x = fp_perlin_noise_2D(scaleCoeff.x, rotCoeff.x, tc+OFF1, 1);
137   nn.y = fp_perlin_noise_2D(scaleCoeff.y, rotCoeff.y, tc+OFF2, 1);
138   nn.z = fp_perlin_noise_2D(scaleCoeff.z, rotCoeff.z, tc+OFF3, 1);
139   nn.w = fp_perlin_noise_2D(scaleCoeff.w, rotCoeff.w, tc+OFF4, 1);
140   n += timeWeights.y*nn;
141
142 #ifdef HQ_INTERPOLATION
143   tc00 = tex2D(texCoords3Tex, (ci+float2(-0.5,-0.5))*GRID_TEXEL_SIZE).xy;
144   tc01 = tex2D(texCoords3Tex, (ci+float2(-0.5, 0.5))*GRID_TEXEL_SIZE).xy;
145   tc10 = tex2D(texCoords3Tex, (ci+float2( 0.5,-0.5))*GRID_TEXEL_SIZE).xy;
146   tc11 = tex2D(texCoords3Tex, (ci+float2( 0.5, 0.5))*GRID_TEXEL_SIZE).xy;
147   tc = lerp(lerp(tc00, tc10, cf.x), lerp(tc01, tc11, cf.x), cf.y);
148 #else
149   tc = tex2D(texCoords3Tex, uv).xy;
150 #endif
151   nn.x = fp_perlin_noise_2D(scaleCoeff.x, rotCoeff.x, tc+OFF1, 2);
152   nn.y = fp_perlin_noise_2D(scaleCoeff.y, rotCoeff.y, tc+OFF2, 2);
153   nn.z = fp_perlin_noise_2D(scaleCoeff.z, rotCoeff.z, tc+OFF3, 2);
154   nn.w = fp_perlin_noise_2D(scaleCoeff.w, rotCoeff.w, tc+OFF4, 2);
155   n += timeWeights.z*nn;
156   return n;
157 }
158
159 // defaults : noiseOffset = noiseFactor = 0.25 => remaps noise to [0,1/2]
160 float fp_flownoise_2D(float2 uv)
161 {
162   half4 n = fp_raw_flownoise_2D(uv);
163   float noise = noiseOffset + noiseFactor*clamp(dot(MOD_N, scaleWeights), -1, 1);
164   float dens = clamp(tex2D(densTex, uv).r-0.5, -1, maximumDensity);;
165   return dens + noise;
166 }
167
168 half4 fp_flownoise_basic(float2 uv : TEXCOORD0) : COLOR
169 {
170   half noise = saturate(fp_flownoise_2D(uv));
171   return half4(noise,noise,noise,1);
172 }
173
```

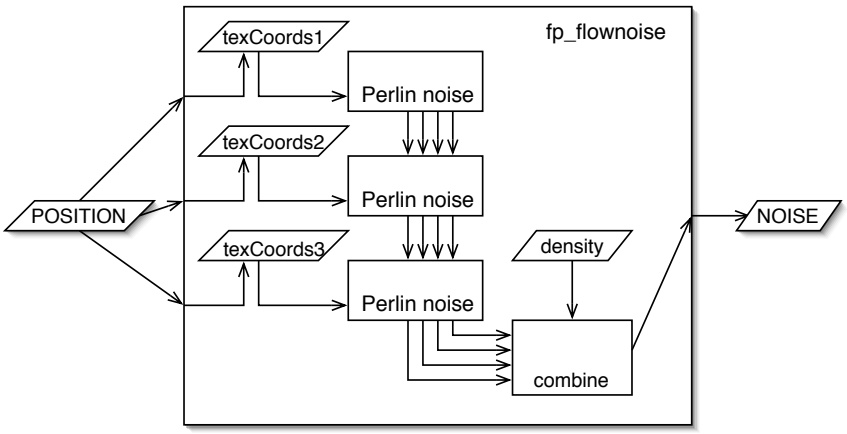

FIG. 23 – Schéma fonctionnel du shader

```
174 technique t_flownoise {
175   pass P0 {
176     VertexProgram = NULL;
177     FragmentProgram = compile fp40 fp_flownoise_basic();
178   }
179 }
```

A.2 Compilation

Le code se compile grâce à la commande suivante :

```
1 cgc -fx -entry fp_flownoise_basic -profile fp40                \
2     -DMOD_N=abs\(n\)-0.15 -DDECORR_TEXEL_SIZE=0.015625         \
3     -DGRID_TEXEL_SIZE=0.015625 -DNOISE_SCALE=0.0625            \
4     -DNOISE_ROTATION=1,2,4,8 -DROTATION_FACTOR=0.25            \
5     -DHQ_INTERPOLATION -DOFFSET=0.92 flownoise.cg
```

On obtient les statistiques suivantes : le shader utilise 763 instructions, 20 registres 32 bits (float) et 14 registres 16 bits (half). Les instructions se répartissent de la manière suivante :

ADDH	13	ADDR	167	ADDR_SAT	1
DP4H	1	FLRR	13	FRCR	12
MADH	12	MADR	48	MAXR	2
MINR	2	MOVH	73	MOVR	36
MOVR_SAT	12	MOVXC	12	MULR	178
COSH	48	SINH	48	SNEHC	12
TEX	73				

Nous avons analysé ce tableau au paragraphe 3.3.1.

A.3 Documentation

A.3.1 Paramètres CG

Les paramètres CG suivants doivent être définis lors l'appel du shader :
- timeWeights : opacité de chacune des couches,
- scaleWeights : poids des différentes échelles de bruit,
- noiseOffset , noiseFactor : le bruit obtenu est linéairement transformé par : noise = noiseFactor ∗ noise + noise Offset,
- maximumDensity : la densité est plafonnée à cette valeur avant qu'on lui additionne le bruit,
- decorrTex : texture de valeurs aléatoires qui sert à dé-corréler spatialement les ondelettes, de taille $N_d \times N_d$,
- densTex : texture qui donne la densité, de taille $N \times N$; seule la première composante est utilisée,
- rotTex : texture dont la i^{e} composante donne la vitesse de rotation dans la i^{e} couche, $i \in \{1, 2, 3\}$, de taille $N \times N$,
- texCoordsiTex : texture qui donne les coordonnées textures directes et inverses dans la i^{e} couche, $i \in \{1, 2, 3\}$, de taille $N \times N$.

A.3.2 Defines

La ligne de compilation du shader doit contenir les *defines* suivants, choisis pour effectuer le maximum de calculs à la compilation :
- MOD_N=f(n) : transformation appliquée au vecteur de bruit n,
- DECORR_TEXEL_SIZE=f : $f = 1/N_d$, pas de la texture de décorrélation,
- GRID_TEXEL_SIZE=f : $f = 1/N$, pas de la grille principale,
- NOISE_SCALE=f : $f = k/N$, échelle fondamentale du flow-noise ; on peut choisir k différent de 1 pour commencer le flow-noise à une échelle différente du pas de la grille principale,
- NOISE_ROTATION=f0,f1,f2,f3 : coefficients contrôlant la vitesse de rotation aux différentes échelles,
- ROTATION_FACTOR=f : multiplicateur appliqué aux coefficients précédents, pour moduler globalement l'activité du fluide,
- HQ_INTERPOLATION : argument optionnel, qui active l'interpolation 32 bits des coordonnées textures, améliorant nettement l'aspect du bruit sous de forts agrandissements,
- OFFSET=f : contrôle le déplacement des grilles aux différentes échelles, lorsqu'on les décale légèrement pour minimiser les artefacts d'alignement.

Références

[Bev] Jason Bevins : Libnoise example: procedural textures. `http://libnoise.sourceforge.net/examples/textures/`.

[Cg] Cg. `http://developer.nvidia.com/Cg`.

[Cho94] Alexandre J. Chorin : *Vorticity and Turbulence*, volume 103 de *Applied Mathematical sciences*. Springer-Verlas, 1994.

[EMP+03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin et Steven Worley : *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers, 2003.

[FK03] Randima Fernando et Mark J. Kilgard : *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FLF] Prof. Glenn R. Flierl, Dr. Sonya Legg et Prof. Raffaele Ferrari : Turbulent Motion in the Atmoshpere and Oceans. `http://ocw.mit.edu/`. Lecture notes published on the MIT OpenCourseWare.

[Lef] Sylvain Lefvebre : LibSL. `http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/`.

[LQV] libQGLViewer. `http://artis.imag.fr/~Gilles.Debunne/QGLViewer/`.

[Ney03] Fabrice Neyret : Advected Textures. *In ACM-SIGGRAPH/EG Symposium on Computer Animation (SCA)*, july 2003.

[OGL] OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org/`.

[Pera] Ken Perlin : Improved Noise reference implementation. `http://mrl.nyu.edu/~perlin/noise/`.

[Perb] Ken Perlin : Noise reference implementation. `http://mrl.nyu.edu/projects/texture/noise.html`.

[Per85] Ken Perlin : An image synthesizer. *In SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM Press.

[Per02] Ken Perlin : Improving noise. *In SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM Press.

[PH89] Ken Perlin et Eric M. Hoffert : Hypertexture. *In* Jeffrey Lane, éditeur : *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23(3), pages 253–262, juillet 1989.

[PN01] Ken Perlin et Fabrice Neyret : Flow Noise. *In Siggraph Technical Sketches and Applications*, page 187, Aug 2001.

[Qt] Qt Homepage – Trolltech. `http://www.trolltech.com/products/qt`.

[SRF05] Andrew Selle, Nick Rasmussen et Ronald Fedkiw : A vortex particle method for smoke, water and explosions. *SIGGRAPH - ACM Trans. Graph.*, 24(3):910–914, 2005.

[Sta99] Jos Stam : Stable Fluids. *In* Alyn Rockwood, éditeur : *Proceedings of the Conference on Computer Graphics (Siggraph99)*, pages 121–128, N.Y., August 8–13 1999. ACM Press.

[Sta01] Jos Stam : A Simple Fluid Solver Based on the FFT. *Journal of Graphics Tools*, 6(2):43–52, 2001.

[Sta03] Jos Stam : Real-Time Fluid Dynamics for Games. *In Proceedings of the Game Developer Conference*, march 2003.

[SU94] John Steinhoff et D. Underhill : Modification of the Euler Equations for « Vorticity Confinement » : Application to the Computation of Interacting Vortex Rings. *Physics of Fluids*, 6(8):2738–2743, 1994.